Documentation Requirements

CS 2334 ~ Fall 2007

These requirements are a mutation of the official Java coding conventions and are more appropriate to an academic environment than the official conventions. The official Java coding conventions are on-line at http://java.sun.com/docs/codeconv/.

All Java source code files should have an extension of **.java**. All Java bytecode files should have an extension of **.class**. Each file should contain only one class or interface, and the name of the file should match the name of the class or interface. For example, a class called **Sample** should be stored in a file named **Sample.java**.

1. Ordering

The ordering of components within a file is as follows:

1. Package statement *(optional)*
2. Import statement(s) *(optional, but almost always necessary)*
3. Javadoc style class header comment
4. Class header
5. Static variables (sorted in the order of public, protected, default, then private variables)
6. Instance variables (sorted in the order of public, protected, default, then private variables)
7. Constructors
8. Methods (sorted in the order of accessors and then mutators)

2. Comments

Java programs have two types of comments. Documentation comments, which are of interest to someone using the class, and implementation comments, which are of interest to someone writing or maintaining the class.

2.1 Documentation Comments

Documentation comments record the behavior, usage, and specifications of a class. In this course we will also use documentation comments to record the design of a class. Documentation comments are in the form of Javadoc comments. Javadoc is a tool that processes Java code files extracting the comments and creating HTML files that present the information contained in the comments in an easy to read and user friendly form. Javadoc comments appear before the code that they describe. Javadoc comes as a part of the Java SDK.

At the beginning of each class, a *Javadoc style class header comment* like the one below must be given. This comment must include the project number, course number (CS 2334), class section, date, and **@author** and **@version** Javadoc tags.

Multi-line Javadoc comments begin with '/**' on a line by itself and a '*' followed by a space on

each line of the comment with the last line consisting of '*/' on a line by itself. A single line Javadoc comment starts with '/**' and ends with '*/'.

```
/**
 * Project #5
 * CS 2334, Section 010
 * May 12, 2004
 * <P>
 * A comment describing the class fully.  This comment may be a page
 * long and use HTML.
 * </P>
 * @author Your name here
 * @version 1.0
 */
```

Note that the package statement and import statement(s) must come before the Javadoc style class header comment. If a package or import statement is placed between the Javadoc style class header comment and the class header itself, then Javadoc will fail to associate the class header comment with the corresponding class.

Each class variable should also be documented as shown in the example below.

```
/** Stores the speed of the airplane as a floating point value. */
float speed;
```

Each method must be preceded by a *method header comment* like the one given below. This comment must give a specification of the purpose and design of the method. The design of the method may include a sequence of steps that must be implemented in the body of the method or pseudo-code for the method.

The *method header comment* must also include the Javadoc tag fields **`@param`**, **`@return`**, and **`@exception`**. Each **`@param`** field specifies the parameters required and provides a short description of its purpose. The **`@return`** field specifies the type of the return value of the method and description of its purpose. Each field specifies the type of any exceptions the method throws and their purpose. Note that your methods may have zero, one, or more parameters and will contain a corresponding number of fields. Additionally, your methods may throw zero, one, or multiple exceptions and will contain a corresponding number of **`@exception`** fields.

Finally, the *method header comment* is used to document the preconditions and postconditions associated with a method. Each precondition and postcondition is specified on a separate line, as shown in the example below, with preconditions specified before the postconditions. Some methods such as accessors that do not change the state of the object or perform any calculations may not need preconditions or postconditions. You must also provide code in your methods that test the preconditions and postconditions. This will be discussed in class. Example code that tests preconditions and postconditions can be found in Section 8.

```
/**
 * A descriptive comment goes here. This comment may be several lines long.
 * <P>
 * Algorithm:<br>
 * 1. Each step of the algorithm is listed here.<br>
 * 2. Be sure to put an html <br> tag after each step so that
 *    each step shows up on a separate line.<br>
 * </P>
 * @param              input      Each paramter has a separate listing like
 *                                this one.
 * @return             Include a descriptive comment describing the return
 *                     type.
 * @exception          IllegalArgumentException    Explain when this exception
 *                                                 will be thrown.
 * <dt><b>Conditions:</b>
 * <dd>PRE  -         List the precondition here. Each precondition has a
 *                    separate listing.
 * <dd>POST -         Postconditions are listed the same way as
 *                    preconditions but with <dd>POST instead of <dd>PRE.
 */
```

It is imperative that you create the Javadoc class header and method header comments using your UML class diagram before you begin programming. This step lets you think about how your classes and methods need to be constructed and implemented at a detailed design level before you become concerned with Java language specific implementation details. First you should create class header comments, class headers, class variables (and their corresponding comments), and Javadoc style method headers and method stubs. Then you can use the Javadoc tool to create a detailed level design specification that is easy to read and will help you as you implement each method and test your program. The command will create a sub-directory named javadocs that contains the Javadoc HTML files.

```
javadoc -breakiterator -private -author -version -d javadocs <source files>
```

This is a very complicated command that may be hard to remember and type. Therefore, an options file has been provided for you named docs.opt on the class website. Download this file into the directory that contains your .java source files and run the command as shown below.

```
javadoc @docs.opt <source files>
```

Also, note that the above commands will not produce Javadoc files for your entire project. To correctly produce Javadoc files for the entire project your must list all of your .java files in place of **<source files>** or you must specify *.java in place of <source files>. See the examples below.

```
javadoc @docs.opt SpellChecker.java CustomIO.java Dictionary.java
```

or

```
javadoc @docs.opt *.java
```

Javadoc comments can be used to create fancy documentation and the Javadoc specification includes many features that are not covered in this document. If you want to explore more options see the Javadoc website http://java.sun.com/j2se/javadoc/. An article on how to write Javadoc documentation comments is available at http://java.sun.com/j2se/javadoc/writingdoccomments/.

2.2 Implementation Comments

Implementation comments describe specific implementation details about your code that are helpful in program maintenance, both debugging and enhancement, and incremental implementation. Implementation comments can be written in a block comment form, a single line form, or a trailing form as shown below.

```
/*
 * This is a block comment.
 * Putting asterisks at the beginning of each line makes the comment
 * more noticeable and easier to read.
 */

/* This is a single line comment on a line by itself. */

iAmAnExression = anotherExpression; // This is a trailing comment.
```

Selecting the proper style of comment for the situation is at the discretion of the programmer. Whenever possible all comments should should be proper English and in the form of complete sentences.

3. Indentation and Line Wrapping

- ***Each level of indentation must consist of one tab*** (usually eight spaces).
- ***Indentation must exactly match the block structure of the code***, shown with curly braces below.

An example is given below, showing the proper positioning of curly braces.

```
/**
 * This method demonstrates the proper indentation of code.
 * This method calculates the following step equation
 * <code>yValue = 0 if xValue < 5, 1 otherwise.</code>
 *
 * @param         xValue    Contains the value to be squared.
 * @return        yValue    Calculated as specified above.
 */
public int myMethod( int myParameter )
{
        if( yValue < 5 )
        {
                return 0;
        }
        else
        {
                return 1;
        }
} // end myMethod
```

Different editing programs interpret tabs to represent different number of spaces, therefore you should use spaces instead of tabs. Many text editors can be set to insert spaces instead of tabs.

- ***Lines must be less than 80 characters in length*** (most printers will not print more than 80 characters of ASCII text per line and will cut-off or wrap the text resulting in hard copy printouts that are difficult to read).

- Word wrap should not be used. Instead ***lines should be broken according to the following rules:***
  - Break after a comma.
  - Break before an operator.
  - Align the new line with the beginning of the expression at the same level on the previous line.
  - If the above rules result in unreadable code, then indent one tab instead.

```
SomeMethod( thisIsAReallyLongParamter, andSoIsThisOne, theLineIsLong,
            soTheLastOneGoesHere );
longExpression = thisIsALongMathExpression + andSoIsThisOne +
                 thisOneShouldBeHere;
```

Many programmers use three or four spaces instead of eight spaces for tabs because using eight spaces makes source code lines longer and more likely to line-wrap.

4. Declarations

- ***Each class variable declaration must be on a line by itself, preceded by a class variable comment*** as discussed in Section 2.1.
- ***All other variable declarations should be on a line by themselves, preceded by a comment*** on the previous line or followed by a trailing comment on the same line as shown below.

```
int size; // Store the number of objects in the array.
```

- ***All declarations should occur at the beginning of a blo***ck, except for loop counters that are declared inside a `for` loop.

```
for( int i = 0; i < size; i++ )
{
        int sum = 0; /* This declaration should not occur
                      * after the next statement.
                      */

        sum = sum + (i * size);
}
```

All declared variables should have unique names within a given code segment. The code below for example, is not allowed by convention, although the compiler will accept it.

```
float grade; // Holds the grade of a student in CS 2334.

grade = 10.0;

if( grade < 20.5 )
{
        float grade; // Holds the grade awarded to an examination.
        grade = 5.5;
}
```

5. Statements

- As a general rule, ***each line should contain at most one statement.*** The code below, for example, is not allowed.

```
argc++; x++; // Don't write two statements on the same line.
argc++;
x++; // This statement and the one on the line above are better.
```

- ***Braces should be used around all blocks,*** even if they consist of only one statement.

```
if( x < 0 )
{
        x = x*-1;
} // This block has proper formatting.
```

- ***Return statements should not contain parenthesis***, unless they improve reliability.

```
return x; // This return statement has correct formatting;
return( x ); // Don't use this style of a return statement.
```

✓ The proper indentation for a nested conditional statement is shown below.

```
if( x < 0 )
{
        x = x + 1;
}
else
{
        x = x - 1;
        if( x > 3 )
        {
                x = 2;
        }
}
```

✓ The proper indentation for loops is shown below.

```
// This is a for loop.
for( int i = 0; i < 10; i++ )
```

```
{
        x++; // Do some calculations here.
}
/*
 * The above braces should be here even though there is only a
 * single statement in the body.
 */


/*
 * This is a for loop with an empty body.  These types of for loops
 * are considered poor style and can cause problems, so be careful
 * with them.
 */
for( int i = 0; i < 20; i++ ); /* This is the one case where not using curly
                                * braces is allowed.
                                */
// EMPTY BODY
// This is a while loop.
while( k < 20 )
{
        x++;
} // These curly braces are also necessary.


// This is a do-while loop.
do
{
        x++;
} while( x < 0 );
```

✓ **switch** statements have the following format.

```
switch( i )
{
        case 1:
                x = 0;
                // This case falls through.
        case 2:
                x = x + 2;
                break; // This case does not fall through.
        case 3:
                x = 3;
                break; // This case doesn't fall through.
        default:
                /*If you think you've got all the cases, throwing an
                 * exception here is a good idea.
                 */
                break;
}
```

✓ **try-catch** statements are formatted as shown below.

```
try
```

```
{
      x = 0;
}
catch( Exception e )
{
      x = 1;
}
finallly
{
      x = 2;
}
```

6. Blank Lines (White-space)

A reasonable number of blank lines (i.e., white-space) improves the readability of code. It is neither desirable nor acceptable to use a blank line between each line of code.

- ***Blank lines should always be used in the following circumstances:***
  - Between sections of a source file
  - Between class and interface declarations
  - Between methods
  - Between the local variables of a method and its first statement
  - Before a block or single line comment
  - Between logical sections inside a method

7. Naming Conventions

Package names are always written in all lower case letters. For example, **`java.io`**.

Class and interface names should be nouns in mixed case with the first letter of each internal word capitalized, for example **`ImageSprite`**.

Method names should be verbs, in mixed case with the first letter lowercase and with the first letter of each internal word capitalized, for example **`run()`** and **`runFast()`**.

Additionally, variables are in mixed case with the first letter lowercase and with the first letter of each internal word capitalized. Variable names may not start with an underscore or a dollar sign. One character variable names should be avoided, except for throwaway counter variables. Constants should be upper case with words separated by underscores as shown in the following example. As a general rule, all constants should be named, except for -1, 0, and 1 in loop counters.

```
int final MY_CONSTANT = 3;
```

8. Example

The following pages contain the code of a a sample class that shows the proper formatting and documentation required for this course. The Javadoc HTML output for this class is posted on class

website in the same place as this document.

```
// If there was a package statement, it would go here.

// If any import statements existed they would go here.

/**
 * Project #0
 * CS 2334, Section 010
 * May 12, 2004
 * <P>
 * This is an example class for CS 2334 that shows proper
 * documentation. It is a class that doesn't do much of anything
 * useful, which makes it hard to write a really great comment up
 * here. It is however, properly documented.
 * </P>
 * @author Java Joe
 * @version 3.0
 */
public class MyString
{
        /**
         * A constant that contains the standard number of
         * characters in a string.
         */
        static public int DEFAULT_SIZE = 10;

        /** A private array to store the characters in the string. */
        private char[] data;

        /**
         * The number of characters that are actually stored in the
         * string.
         */
        private int size;


        /**
         * A constructor for an empty string of
         * <code>currentSize</code> elements.
         *
         * @param         currentSize    The number of characters in the
         *                               string.
         * @throws        IllegalArgumentException    if currentSize < 0
         * <dt><b>Conditions:</b>
         * <dd>PRE  -     <code>currentSize</code> contains a non-negative
         *                integer.
         * <dd>POST -     The character array data is of length
         *                <code>currentSize</code>.
         */
        public MyString( int currentSize )
        {
                /* Test preconditions.
                 * In public methods, preconditions are tested with an "if"
```

```
         * statement that throws an IllegalArgumentException if the
         * the condition fails. In private classes, preconditions
         * can be tested using "assert" statements.
         */
        if( currentSize < 0 )
        {
                throw new IllegalArgumentException(
                        "Illegal currentSize: " + currentSize );
        }

        data = new char[currentSize];
        size = 0; // Set the number of characters stored to zero.

        /* Test postconditions.
         * In all methods, postconditions are tested using "assert"
         * statements.
         */
        assert (data.length == currentSize);
} // end MyString

/**
 * An accessor for the variable <code>size</code>.
 *
 * @return        The number of characters in the current string.
 */
public int getLength()
{
        return size;
} // end getLength

/**
 * The accessor for the character array <code>data</code>.
 *
 * @return        A reference to the array <code>data</code>.
 */
public char[] getData()
{
        return data;
} // end getData

/**
 * A mutator that sets the size of the array. This is a simple
 * method, so it does not have a very sophisticated algorithm.
 * <P>
 * Algorithm:<br>
 * 1. Allocate more space.<br>
 * 2. Copy existing data into the new array.<br>
 * 3. Record changes in the instance variables.<br>
 * </P>
 * @param         newLength    The desired length for the modified
 *                             string.
 * @throws        IllegalArgumentException    if newLength < 0
 * <dt><b>Conditions:</b>
 * <dd>PRE  -      <code>newLength</code> contains a non-negative
```

```
 *                integer.
 * <dd>POST -     The character array data is of length
 *                <code>newLength</code>.
 */
public void setLength( int newLength )
{
        /* Test preconditions. */
        if( newLength < 0 )
        {
                throw new IllegalArgumentException(
                        "Illegal newLength: " + newLength );
        }

        /*
         * Contains the number of characters in the string after it
         * has been re-sized.
         */
        int newSize;
        // Allocate more space.
        char[] newData = new char[newLength];


        // Copy existing data into the new array.
        for( int i = 0; i< Math.min( newLength, size ); i++ )
        {
                newData[i] = data[i];
        } // end for


        // Record changes in the instance variables.
        data = newData;
        size = Math.min( newLength, size );

        /* Test postconditions. */
        assert (data.length == newLength);
} // end setLength

/**
 * A mutator that changes the contents of the character array
 * <code>data</code>.
 *
 * @param        newData    A reference to the character array to be
 *                          used.
 * <dt><b>Conditions:</b>
 * <dd>PRE  -    The character array data has the same length as
 *               <code>newData</code>.
 * <dd>POST -    Both data and newData refer to the same character
 *               array.
 */
public void setData( char[] newData )
{
        /*
         * Set data to refer to the character referenced by
         * newData.
```

```
                 */
                data = newData;
                size = data.length;

                /* Test postconditions. */
                assert (data.length == newData.length);
                assert (data == newData);

        } // end setData

} // End of class MyString
```